VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**THE FACULTY OF INFORMATION TECHNOLOGY**



**Design and Analysis of Algorithms**

**Final Report**

**(5 item-sets mining algorithms)**

*Instructor*: **NGUYEN CHI THIEN**

*Students*: **Lê Phước Thịnh – 520K0343**

**Trịnh Bảo Toàn – 520K0332**

**Nguyễn Duy Tuấn – 520K0232**

Class **: 20K50301**

Enrolled course  **: 24**

**HO CHI MINH CITY, 2022**

**ACKNOWLEDGEMENT**

*I would like to express my gratitude and thanks to my Design and Analysis of Algorithms lecturer Mr. Nguyen Chi Thien for his wonderful guidance throughout this course.*

*I am also grateful to my friends for their help, and to all those who contributed directly or indirectly towards the completion of this school project.*

*During this project, I acquired many valuable skills, and I hope that in the years to come, those skills will be put to good use.*

# TABLE OF CONTENTS

[TABLE OF CONTENTS 2](#_Toc121001083)

[A. Frequent Patterns Mining Algorithms: 3](#_Toc121001084)

[1. Apriori Algorithm: 3](#_Toc121001085)

[2. Apriori TID Algorithm: 7](#_Toc121001086)

[3. Eclat Algorithm: a vertical depth-first search algorithm 11](#_Toc121001087)

[4. FP – Growth Algorithm: 15](#_Toc121001088)

[5. H – Mine Algorithm: 28](#_Toc121001089)

[B. Proofs that our algorithm implementations run faster than Philippe Fournier’s program: 36](#_Toc121001090)

[C. Comparation of Time and Space Usage of all mentioned algorithms: 40](#_Toc121001091)

[D. Significance Rubrik: 42](#_Toc121001092)

# A. Frequent Patterns Mining Algorithms:

## 1. Apriori Algorithm:

**a) Input and Output Description:**

This technique requires a transaction database and a user-specified threshold ‘minsup’ (value between 0 to 1, representing 0% to 100% respectively).

Ex: database = [[1,2,3], [1,2], [3,5,1],[3,5,1,2]] (database with 4 transactions, separated by arrays)

minsup = 0.5 (a 50% threshold)

The expected output is to be the most frequent patterns in the form of k-itemsets, with k from 1 to n, n is the number of items in the longest transaction in the database.

Ex: Frequent 1-itemset: [1] [2] [3]

Frequent 2-itemset: [1,2] [3,5]

Frequent 3-itemset: [1,3,5]

Frequent 4-itemset: [1,2,3,5]

**b) Implementation:**

*import* time

*import* tracemalloc

# *The Apiori algorithm*

# *input: D: a set of transactions database, minsup: a user-specified threshold*

# *output: the set of frequent itemsets*

# *Scan the database to calculate the support of all items in I;*

# *F1 = {i|i ∈ I ∧ sup({i}) ≥ minisup }; // F1 : frequent 1-itemsets*

# *k = 2;*

# *while Fk = ∅ do*

# *Ck = CandidateGeneration (Fk - 1) ; // Ck : candidate k-itemsets*

# *Remove each candidate X ∈ Ck that contains a (k - 1)-itemset that is not in Fk-1;*

# *Scan the database to calculate the support of each candidate X ∈ Ck;*

# *Fk = {X|X ∈ Ck ∧ sup(X) ≥ minsup} ; // Fk : frequent k-itemsets*

# *k = k + 1;*

# *end*

# *return Uk = 1...k Fk;*

def *Apiori*(D, minsup): # *D: a set of transactions database, minsup: a user-specified threshold*

I = set() # *I: a set of items*

*for* t *in* D:

*for* i *in* t:

I.add(i)

F1 = set() # *F1 : frequent 1-itemsets*

*for* i *in* I:

*if* sup({i}, D) >= minsup:

F1.add(frozenset({i}))

F = [F1] # *F : a set of frequent itemsets*

k = 2

*while* len(F[k - 2]) > 0:

Ck = CandidateGeneration(F[k - 2]) # *Ck : candidate k-itemsets*

Fk = set() # *Fk : frequent k-itemsets*

*for* X *in* Ck:

*if* sup(X, D) >= minsup:

Fk.add(X)

F.append(Fk)

k += 1

*return* F

def *CandidateGeneration*(Fk\_1): # *Fk\_1 : frequent k-1-itemsets*

Ck = set()

*for* i *in* Fk\_1:

*for* j *in* Fk\_1:

*if* len(i.union(j)) == len(i) + 1:

Ck.add(i.union(j))

*return* Ck

def *sup*(X, D): # *X: a set of items, D: a set of transactions database*

count = 0

*for* t *in* D:

*if* X.issubset(t):

count += 1

*return* count / len(D)

def *main*():

D = [{1, 3, 4}, {2, 3, 5}, {1, 2, 3, 5}, {2, 5}, {1, 2, 3, 5}]

minsup = 0.4

start = time.time()

tracemalloc.start()

F = Apiori(D, minsup)

*for* i *in* range(len(F)-1):

print("Frequent " + str(i + 1) + "-itemsets: " + str(F[i]))

end = time.time()

elapsed = (end - start) \* 1000

usage = tracemalloc.get\_traced\_memory()

total = usage[1] - usage[0]

kb = total /1024

print("Time taken: %f ms" % elapsed)

print(f"memory use in KiB:{kb:.3f}")

*if* \_\_name\_\_ == "\_\_main\_\_":

main()

**c) Algorithm Analysis:**

Apriori’s Time Complexity = O(2^n)

Apriori’s Space Complexity = O(2^n)

With n represents the horizontal width present in the database.

## 2. Apriori TID Algorithm:

**a) Input and Output Description:**

This technique requires a transaction database and a user-specified threshold ‘minsup’ (value between 0 to 1, representing 0% to 100% respectively).

Ex: database = [[1,2,3], [1,2], [3,5,1],[3,5,1,2]] (database with 3 transactions)

minsup = 0.5 (a 50% threshold)

The expected output is to be the most frequent patterns in the form of k-itemsets, with k from 1 to n, n is the number of items in the longest transaction in the database.

Ex: Frequent 1-itemset: [1] [2] [3]

Frequent 2-itemset: [1,2] [3,5]

Frequent 3-itemset: [1,3,5]

Frequent 4-itemset: [1,2,3,5]

**b) Implementation:**

*import* time

*import* tracemalloc

def *Database*(input): # *Function to read the database*

listOfTIDs = []

listOfItems = []

*for* i *in* input:

trans = []

*for* j *in* i:

trans.append([j])

*if* [j] not in listOfItems:

listOfItems.append([j])

listOfTIDs.append(trans)

*return* listOfTIDs, listOfItems

def *countSup*(itemSet, transDB): # *Function to count support of an itemset*

count = 0

*for* tid *in* transDB:

*if* itemSet in tid:

count += 1

*return* count

def *getFrequentItemSets*(C, transDB, minsup): # *Function to generate frequent itemsets*

L = [itemSet *for* itemSet *in* C *if* countSup(itemSet, transDB) >= minsup]

*return* L

def *candidateItemSets*(L): # *Function to generate candidate itemsets*

C = []

*for* i *in* range(len(L)):

*for* j *in* range(i+1, len(L)):

*if* L[i][:-1] == L[j][:-1]:

C.append(sorted(set(L[i]).union(set(L[j]))))

*return* C

def *getPassC*(prevPassC, C): # *Function to generate candidate itemsets for passing*

passC = []

*if* (C):

k = len(C[0]) - 1 # *last index*

*for* t *in* range(len(prevPassC)):

Ct= []

*for* c *in* C:

a = c[:-1]

b = c[:k-1] + c[k:]

*if* a in prevPassC[t] and b in prevPassC[t]:

Ct.append(c)

*if* Ct: passC.append(Ct)

*return* passC

def *aprioriTID*(database, minsup): # *Function to generate frequent itemsets*

C\_, C = Database(database)

L = getFrequentItemSets(C, C\_, 3)

res = []

*while*(L):

res.append(L)

C = candidateItemSets(L)

C\_= getPassC(C\_, C)

L = getFrequentItemSets(C, C\_, minsup)

*return* res

def *main*():

test\_data = [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5], [1, 2, 3, 5]]

start = time.time()

tracemalloc.start()

frequent\_items = aprioriTID(test\_data, 2)

*for* i *in* range(len(frequent\_items)):

print('Frequent', f'{i+1}-itemset: {frequent\_items[i]}')

end = time.time()

elapsed = (end - start) \* 1000

usage = tracemalloc.get\_traced\_memory()

total = usage[1] - usage[0]

kb = total /1024

print("Time taken: %f ms" % elapsed)

print(f"memory use in KiB:{kb:.3f}")

*if* \_\_name\_\_ == "\_\_main\_\_":

main()

**c) Algorithm Analysis:**

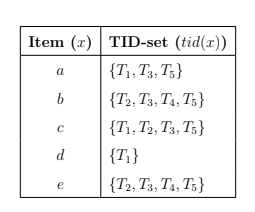
Apriori’s Time Complexity = O(2^n)

Apriori’s Space Complexity = O(2^n)

With n represents the horizontal width present in the database.

## 3. Eclat Algorithm: a vertical depth-first search algorithm

**a) Input and Output Description:**

****

* input: R a set of items with their tidsets, minsup: a user threshold
* output: itemset X ∈ R such that |tid(X)| ≥ minsup do

**b) Implementation:**

***import* time**

***import* tracemalloc**

**true\_res = []**

**def *\_list*(i,j):**

**temp\_list = []**

**temp\_list.append(i)**

**temp\_list.append(j)**

**final\_list = []**

***for* x *in* temp\_list:**

***for* x\_ *in* x:**

***if* x\_ not in final\_list:**

**final\_list.append(x\_)**

***return* final\_list**

**def *sorted\_res*(list,max\_length):**

**my\_list = []**

**true\_list = []**

***for* x *in* list:**

**my\_list.append(sorted(x))**

***for* x *in* my\_list:**

***if* x not in true\_list and len(x)==max\_length+1:**

**true\_list.append(sorted(x))**

***return* true\_list**

**def *printRes*(res):**

***for* i *in* res:**

**print(i)**

**def *eclat*(R, minsup):**

***if* R=={}:**

**#*printRes(true\_res)***

***return***

**res = []**

***for* i *in* R:**

**max\_length = (len(i))**

***if* max\_length==1:**

**itemset\_1 = []**

***for* i *in* R:**

***if* len(R[i])>=minsup:**

**itemset\_1.append([i])**

**true\_res.append(itemset\_1)**

**check\_repeat = []**

**R\_ = {}**

***for* i *in* R.copy():**

***for* j *in* R.copy():**

***if* i != j:**

**temp\_set = R[i]&R[j]**

***if* len(temp\_set) >=minsup and ([i,j]) not in check\_repeat:**

**final\_list = \_list(i,j)**

**final\_list = sorted(final\_list)**

**res.append(final\_list)**

***if*(len(tuple(final\_list)))==max\_length+1:**

**R\_[tuple(final\_list)] = {st *for* st *in* temp\_set}**

**check\_repeat.append([i,j])**

**check\_repeat.append([j,i])**

**res = sorted\_res(res,max\_length)**

**true\_res.append(res)**

**eclat(R\_,minsup)**

**def *main*():**

**"""Main function."""**

**R = {'1': {'t1', 't3', 't5'}, '2': {'t2', 't3', 't4', 't5'} , '3': {'t1', 't2', 't3', 't5'}, '4': {'t1'}, '5': {'t2', 't3', 't4', 't5'}}**

**minsup = 2**

**start = time.time()**

**tracemalloc.start()**

**eclat(R, minsup)**

***for* i *in* range(len(true\_res)-1):**

**print('Frequent', f'{i+1}-itemset: {true\_res[i]}')**

**end = time.time()**

**elapsed = end - start**

**usage = tracemalloc.get\_traced\_memory()**

**total = usage[1] - usage[0]**

**kb = total /1024**

**usage = tracemalloc.get\_traced\_memory()**

**print("Time taken: %f ms" % elapsed)**

**print(f"memory use in KiB:{kb:.3f}")**

***if* \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**c) Algorithm Analysis:**

Eclat’s Time Complexity = Time Complexity for Initial Operations + Time Complexity for Searching Pattern = **O(m \* z)** **+** **O(m \* n)** **+ O**(**2^n -1**) \* **O(m)**

Eclat’s Space Complexity = Space Complexity for Initial Operations + Space Complexity for Searching Pattern = **O(m \* n)** + **O**(**2^n -1**) **\*** **O(m)**

## 4. FP – Growth Algorithm:

**a) Input and Output Description:**

input : a transaction database

minsup: a user-specified threshold

output: the set of frequent itemsets

**b) Implementation:**

***import* time**

***import* tracemalloc**

**class FPTreeNode():**

**def \_\_init\_\_(self, item=None, support=1):**

**# *'Value' of the item***

**self.item = item**

**# *Number of times the item occurs in a***

**# *transaction***

**self.support = support**

**# *Child nodes in the FP Growth Tree***

**self.children = {}**

**class FPGrowth():**

**"""A method for determining frequent itemsets in a transactional database.**

**This is done by building a so called FP Growth tree, which can then be mined**

**to collect the frequent itemsets. More effective than Apriori for large transactional**

**databases.**

**Parameters:**

**-----------**

**min\_sup: float**

**The minimum fraction of transactions an itemets needs to**

**occur in to be deemed frequent**

**"""**

**def \_\_init\_\_(self, min\_sup=0.3):**

**self.min\_sup = min\_sup**

**# *The root of the initial FP Growth Tree***

**self.tree\_root = None**

**# *Prefixes of itemsets in the FP Growth Tree***

**self.prefixes = {}**

**self.frequent\_itemsets = []**

**# *Count the number of transactions that contains item.***

**def *\_calculate\_support*(self, item, transactions):**

**count = 0**

***for* transaction *in* transactions:**

***if* item in transaction:**

**count += 1**

**support = count**

***return* support**

**def *\_get\_frequent\_items*(self, transactions):**

**""" Returns a set of frequent items. An item is determined to**

**be frequent if there are atleast min\_sup transactions that contains**

**it. """**

**# *Get all unique items in the transactions***

**unique\_items = set(**

**item *for* transaction *in* transactions *for* item *in* transaction)**

**items = []**

***for* item *in* unique\_items:**

**sup = self.\_calculate\_support(item, transactions)**

***if* sup >= self.min\_sup:**

**items.append([item, sup])**

**# *Sort by support - Highest to lowest***

**items.sort(key=lambda item: item[1], reverse=True)**

**frequent\_items = [[el[0]] *for* el *in* items]**

**# *Only return the items***

***return* frequent\_items**

**def *\_insert\_tree*(self, node, children):**

**""" Recursive method which adds nodes to the tree. """**

***if* not children:**

***return***

**# *Create new node as the first item in children list***

**child\_item = children[0]**

**child = FPTreeNode(item=child\_item)**

**# *If parent already contains item => increase the support***

***if* child\_item in node.children:**

**node.children[child.item].support += 1**

***else*:**

**node.children[child.item] = child**

**# *Execute \_insert\_tree on the rest of the children list***

**# *from the new node***

**self.\_insert\_tree(node.children[child.item], children[1:])**

**def *\_construct\_tree*(self, transactions, frequent\_items=None):**

***if* not frequent\_items:**

**# *Get frequent items sorted by support***

**frequent\_items = self.\_get\_frequent\_items(transactions)**

**unique\_frequent\_items = list(**

**set(item *for* itemset *in* frequent\_items *for* item *in* itemset))**

**# *Construct the root of the FP Growth tree***

**root = FPTreeNode()**

***for* transaction *in* transactions:**

**# *Remove items that are not frequent according to***

**# *unique\_frequent\_items***

**transaction = [item *for* item *in* transaction *if* item in unique\_frequent\_items]**

**transaction.sort(key=lambda item: frequent\_items.index([item]))**

**self.\_insert\_tree(root, transaction)**

***return* root**

**def *\_is\_prefix*(self, itemset, node):**

**""" Makes sure that the first item in itemset is a child of node**

**and that every following item in itemset is reachable via that path """**

***for* item *in* itemset:**

***if* not item in node.children:**

***return* False**

**node = node.children[item]**

***return* True**

**def *\_determine\_prefixes*(self, itemset, node, prefixes=None):**

**""" Recursive method that adds prefixes to the itemset by traversing the**

**FP Growth Tree"""**

***if* not prefixes:**

**prefixes = []**

**# *If the current node is a prefix to the itemset***

**# *add the current prefixes value as prefix to the itemset***

***if* self.\_is\_prefix(itemset, node):**

**itemset\_key = self.\_get\_itemset\_key(itemset)**

***if* not itemset\_key in self.prefixes:**

**self.prefixes[itemset\_key] = []**

**self.prefixes[itemset\_key] += [{"prefix": prefixes, "support": node.children[itemset[0]].support}]**

***for* child\_key *in* node.children:**

**child = node.children[child\_key]**

**# *Recursive call with child as new node. Add the child item as potential***

**# *prefix.***

**self.\_determine\_prefixes(itemset, child, prefixes + [child.item])**

**def *\_get\_itemset\_key*(self, itemset):**

**""" Determines the look of the hashmap key for self.prefixes**

**List of more strings than one gets joined by '-' """**

***if* len(itemset) > 1:**

**itemset\_key = "-".join(itemset)**

***else*:**

**itemset\_key = str(itemset[0])**

***return* itemset\_key**

**def *\_determine\_frequent\_itemsets*(self, conditional\_database, suffix):**

**# *Calculate new frequent items from the conditional database***

**# *of suffix***

**frequent\_items = self.\_get\_frequent\_items(conditional\_database)**

**cond\_tree = None**

***if* suffix:**

**cond\_tree = self.\_construct\_tree(conditional\_database, frequent\_items)**

**# *Output new frequent itemset as the suffix added to the frequent***

**# *items***

**self.frequent\_itemsets += [el + suffix *for* el *in* frequent\_items]**

**# *Find larger frequent itemset by finding prefixes***

**# *of the frequent items in the FP Growth Tree for the conditional***

**# *database.***

**self.prefixes = {}**

***for* itemset *in* frequent\_items:**

**# *If no suffix (first run)***

***if* not cond\_tree:**

**cond\_tree = self.tree\_root**

**# *Determine prefixes to itemset***

**self.\_determine\_prefixes(itemset, cond\_tree)**

**conditional\_database = []**

**itemset\_key = self.\_get\_itemset\_key(itemset)**

**# *Build new conditional database***

***if* itemset\_key in self.prefixes:**

***for* el *in* self.prefixes[itemset\_key]:**

**# *If support = 4 => add 4 of the corresponding prefix set***

***for* \_ *in* range(el["support"]):**

**conditional\_database.append(el["prefix"])**

**# *Create new suffix***

**new\_suffix = itemset + suffix *if* suffix *else* itemset**

**self.\_determine\_frequent\_itemsets(conditional\_database, suffix=new\_suffix)**

**def *find\_frequent\_itemsets*(self, transactions, suffix=None, show\_tree=False):**

**self.transactions = transactions**

**# *Build the FP Growth Tree***

**self.tree\_root = self.\_construct\_tree(transactions)**

**self.\_determine\_frequent\_itemsets(transactions, suffix=None)**

***return* self.frequent\_itemsets**

**def *main*():**

**transactions = [**

**[1, 3, 4],**

**[2, 3, 5],**

**[1, 2, 3, 5],**

**[2, 5],**

**[1, 2, 3, 5]**

**]**

**start = time.time()**

**tracemalloc.start()**

**fpg = FPGrowth(min\_sup=0.4)**

**frequent\_itemsets = fpg.find\_frequent\_itemsets(transactions, show\_tree=True)**

**print("Frequent itemsets: ")**

***for* itemsets *in* frequent\_itemsets:**

**print("\t",itemsets)**

**end = time.time()**

**elapsed = (end - start) \* 1000**

**usage = tracemalloc.get\_traced\_memory()**

**total = usage[1] - usage[0]**

**kb = total /1024**

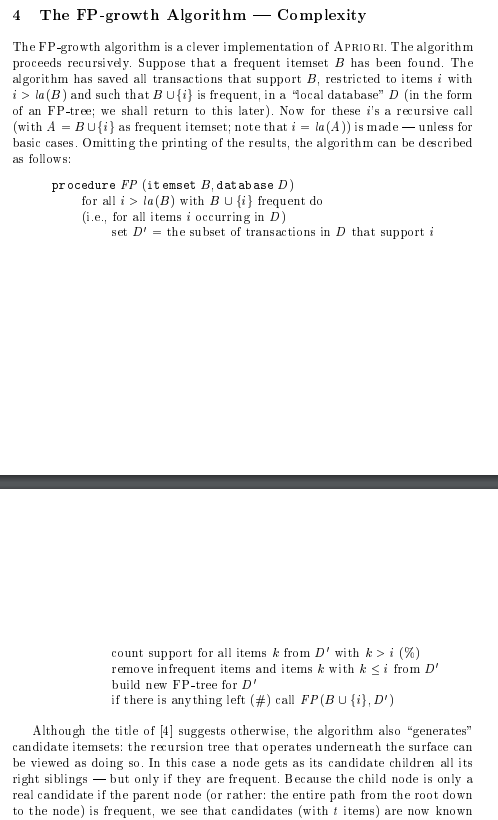
**print("Time taken: %f ms" % elapsed)**

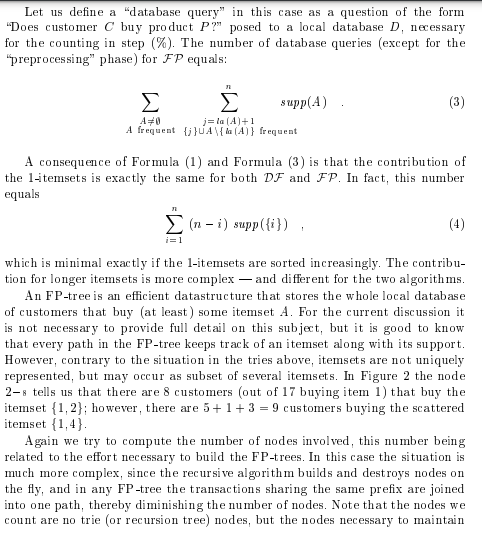
**print(f"memory use in KiB:{kb:.3f}")**

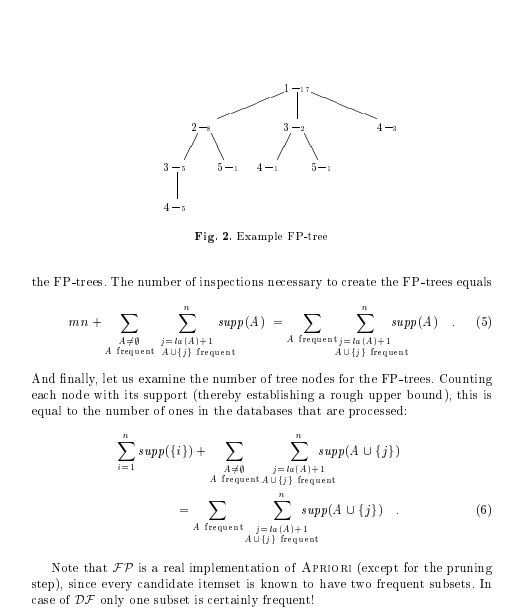
***if* \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**c) Algorithm Analysis:**







## 5. H – Mine Algorithm:

**a) Input and Output Description:**

* input : a transaction database, minsup: a user-specified threshold
* output: the set of frequent itemsets

**b) Implementation:**

***import* math**

***import* time**

***import* tracemalloc**

**def *runHmine*(datalist, min\_support):**

**transactionCount = len(datalist) #*Total Transaction Count in the Input dataset***

**# *The minimum support is calculated using the ceil function.***

**# *min\_support which is passed to this algorithm is in precentage(%). This is converted to number of transactions.***

**minSupport = math.ceil(min\_support \* transactionCount)**

**out\_frequents\_items = [] # *Frequent Items generated by H-Mine are appended one by one to this list***

**itemsetBuffer = [None] \* 200 # *This buffer variable is used to store the Items under recursion to generate the final frequent-itemset***

**mapItemToSupport = {} #*This Dictionary will be used to store the support values of all the unique items in input dataset***

**mapItemRow = {} #*This Dict will help us build the H-Struct table.***

**# *This function is used to generate frequent items using values in itemsetBuffer and prefixlen and apped that value to output list out\_frequents\_items***

**def *writeOut* (prefix, prefixlen, item, support):**

**freq = []**

***for* val *in* range(prefixlen):**

**freq.append(prefix[val])**

**freq.append(item)**

**out\_str = ""**

***for* val *in* freq:**

**out\_str += val + ","**

**temp = f'{out\_str[:-1]} # Supp: {support}' #*Temp has the frequent itemset for current iteration***

**out\_frequents\_items.append(temp) #*Appending temp values to Output list out\_frequents\_items***

**# *Creating Class called Row to store the itemsets objects in form item, support of item, item pointer***

**class Row:**

**def \_\_init\_\_(self, item):**

**self.item = item**

**self.support = 0**

**self.pointer = []**

**#*Building mapItemToSupport Dictionary with Unique Items in input dataset and it's support value***

***for* tran *in* datalist:**

***for* item *in* tran:**

***if* item not in mapItemToSupport.keys():**

**mapItemToSupport[item] = 1**

***else* :**

**mapItemToSupport[item] += 1**

**rowlist=[] #*This acts as a Header table storing the list of row objects***

***for* keys *in* mapItemToSupport.keys():**

***if* mapItemToSupport[keys] >= minSupport:**

**rowItem = Row(keys);**

**rowItem.support = mapItemToSupport[keys]**

**rowlist.append(rowItem)**

**mapItemRow[keys] = rowItem**

**# *This is used to find the frequent-item cell of the database.***

**# *All the other frequent-itemsets will be found using this cell.***

**cell = [] #*This variable stores all the frequent projections in all the transactions seperated by -1***

**flist = sorted(mapItemRow.keys()) # *f-list of frequent itemsets in sorted order***

**idx = 0**

**#*This loop is used to append frequent projections in Cell list and append their respective pointers in mapItemRow dictionary***

***for* tran *in* datalist:**

**temp=[]**

***for* item *in* tran:**

***if* item in flist:**

***if* item not in temp: # *checking double occurences in the list***

**temp.append(item)**

***if* temp:**

**temp = sorted(temp, key = lambda x: mapItemToSupport[x])**

***for* x *in* temp:**

**cell.append(x)**

**mapItemRow[x].pointer.append(idx)**

**idx += 1**

**cell.append(-1)**

**idx += 1**

**# *This function impletements this logic for H-mine algorithm and is called recursively***

**def *hmine*(prefix=[], prefixlen=0, rowlist=[]):**

***for* row *in* rowlist: #*Traversing the header table (rowlist)***

**newRowlist=[]**

**mapItemRow.clear()**

**#*traversing all pointers of row object in row list and building new recursive sub-level header***

***for* pointer *in* row.pointer:**

**pointer+=1**

***if* cell[pointer]==-1:**

***continue*;**

**#*Generating the row objects and incresing the support for all the unique items in row objects***

***while* cell[pointer] != -1 :**

**item=cell[pointer]**

***if* mapItemRow.get(item,None) == None :**

**rowItem = Row(item)**

**rowItem.support = 1**

**rowItem.pointer.append(pointer)**

**mapItemRow[item] = rowItem**

***else*:**

**mapItemRow[item].support += 1**

**mapItemRow[item].pointer.append(pointer)**

**pointer += 1**

**#*Appending only those row objects which have support greater than min\_support***

***for* entry *in* mapItemRow:**

**currentRow = mapItemRow[entry]**

***if* currentRow.support >= minSupport:**

**newRowlist.append(currentRow)**

**#*Calling writeOut function to generate the frequent items and store in output list***

**writeOut(itemsetBuffer, prefixlen, row.item, row.support)**

**#*Sorting newRowlist in lexical order***

***if* len(newRowlist) != 0 :**

**newRowlist = sorted(newRowlist, key = lambda x : x.support)**

**#*Store current row item in buffer before recursion so that it can be used to build the frequent itemset values***

**itemsetBuffer[prefixlen] = row.item**

**hmine(prefix, prefixlen+1, newRowlist) #*recursively calling Hmine algorithm***

**hmine(itemsetBuffer, 0, rowlist) #*Calling Hmine algorithm for first time using empty Buffer and 0 as prefixlength and initial value of rowlist Header.***

**print(f'\nNumber of transactions: {len(datalist)}')**

**print('\nFrequent items found in database:\n')**

***for* ans *in* sorted(out\_frequents\_items):**

**print(ans)**

***if* \_\_name\_\_ == '\_\_main\_\_':**

**test\_data = [['1', '3', '4'], ['2', '3', '5'], ['1', '2', '3', '5'], ['2', '5'], ['1', '2', '3', '5']]**

**start = time.time() #*noting the starting time before calling algorithm***

**tracemalloc.start()**

**runHmine(test\_data, 0.4)**

**end = time.time() #*noting the ending time after completion of algorithm***

**elapsed = (end - start) \* 1000**

**usage = tracemalloc.get\_traced\_memory()**

**total = usage[1] - usage[0]**

**kb = total /1024**

**print("Time taken: %f ms" % elapsed)**

**print(f"memory use in KiB:{kb:.3f}")**

**c) Algorithm Analysis:**

H-mine’s time complexity: O(n^2)

With n is the number of unique items in the database.

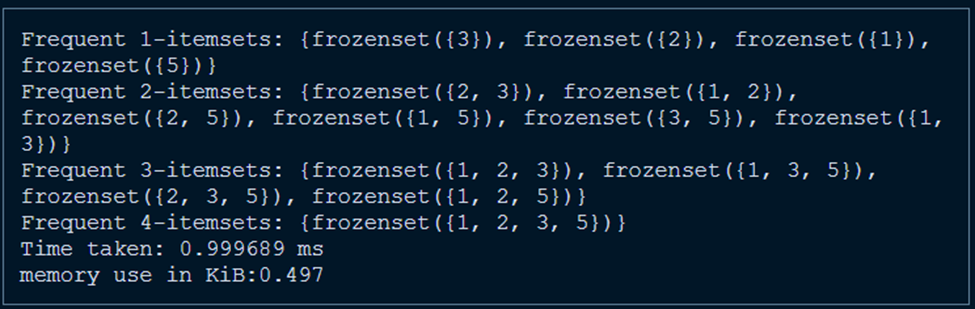
# B. Proofs that our algorithm implementations run faster than Philippe Fournier’s program:

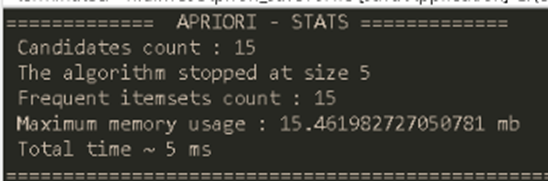
* *Result in Blue color: Ours algorithm implementations*
* *Result in Yellow color: Philippe Fournier’s software*

*Both use the same database of 5 transactions.*

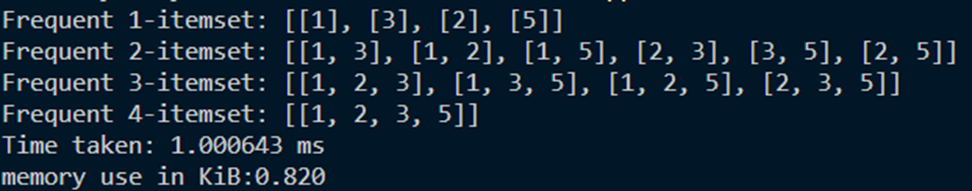
**Database = [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5], [1, 2, 3, 5]]**

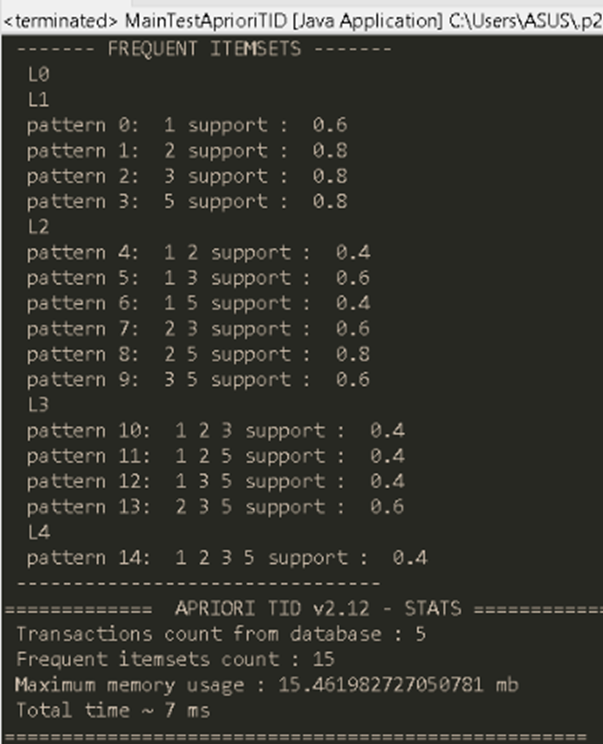
1. **Apriori algorithm:**

****

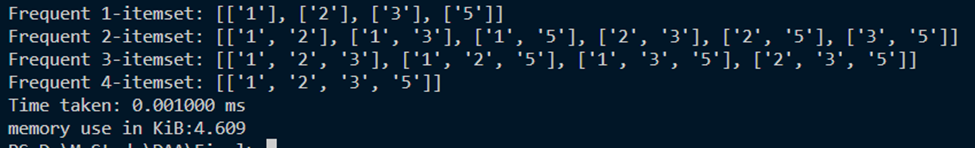
****

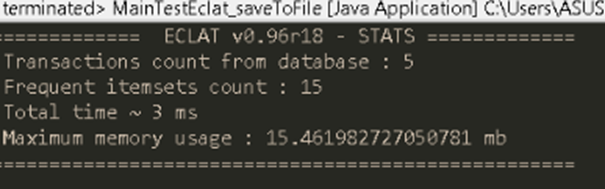
1. **Apriori TID-list algorithm:**

****

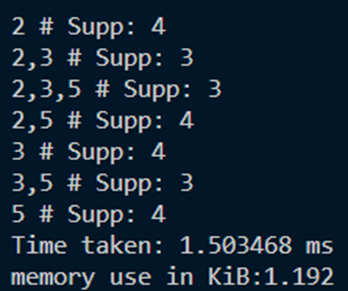
****

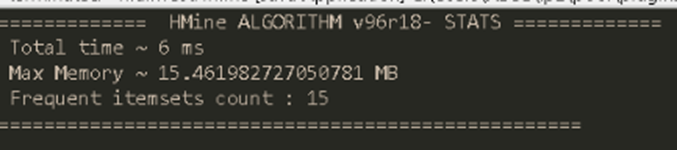
1. **ECLAT Algorithm:**

****

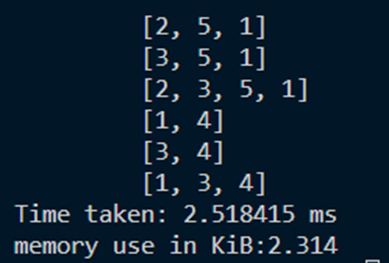
****

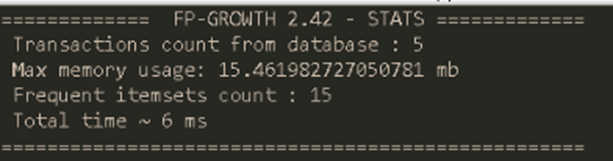
1. **H-Mine Algorithm:**

****

****

1. **FP - Growth Algorithm:**

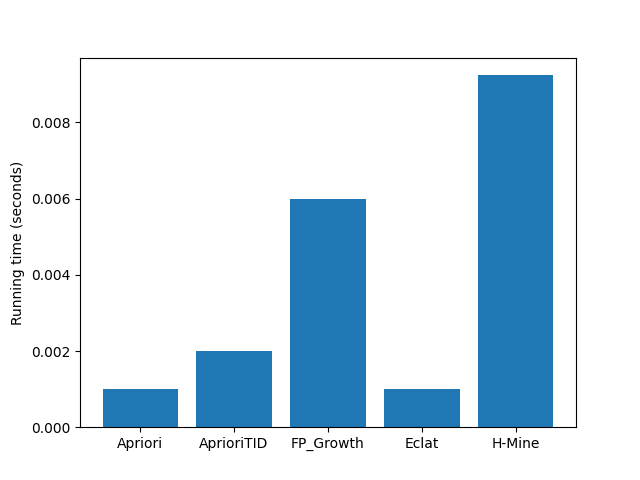
****

****

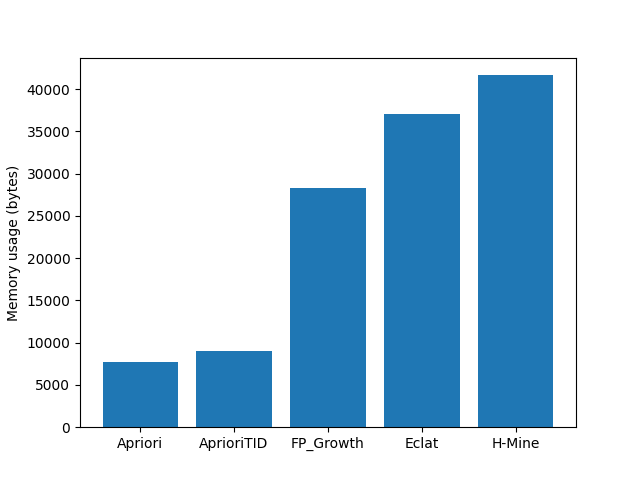
# C. Comparation of Time and Space Usage of all mentioned algorithms:

The database used in all of these samples is the same as the database used above.

*a) Time Usage:*

****

*b) Memory Usage:*

**

# D. Significance Rubrik:

|  |  |
| --- | --- |
| **Name** | **Significance in Contribution** |
| Trinh Bao Toan | 1.0 |
| Le Phuoc Thinh | 0.75 |
| Nguyen Duy Tuan | 0.75 |

**-----End of Report-----**